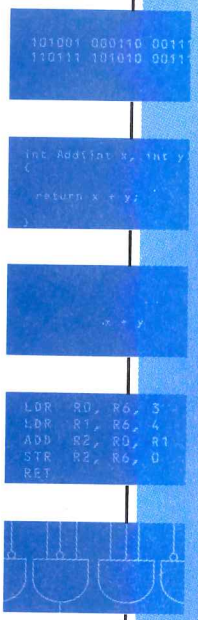
chapter

10



And, Finally ... The Stack

We have finished our treatment of the LC-3 ISA. Before moving up another level of abstraction in Chapter 11 to programming in C, there is a particularly important fundamental topic that we should spend some time on: the *stack*. First we will explain in detail its basic structure. Then, we will describe three uses of the stack: (1) interrupt-driven I/O—the rest of the mechanism that we promised in Section 8.5, (2) a mechanism for performing arithmetic where the temporary storage for intermediate results is a stack instead of general purpose registers, and (3) algorithms for converting integers between 2's complement binary and ASCII character strings. These three examples are just the tip of the iceberg. You will find that the stack has enormous use in much of what you do in computer science and engineering. We suspect you will be discovering new uses for stacks long after this book is just a pleasant memory.

We will close our introduction to the ISA level with the design of a calculator, a comprehensive application that makes use of many of the topics studied in this chapter.

10.1 The Stack: Its Basic Structure

10.1.1 The Stack—An Abstract Data Type

Throughout your future usage (or design) of computers, you will encounter the storage mechanism known as a *stack*. Stacks can be implemented in many different ways, and we will get to that momentarily. But first, it is important to know that the concept of a stack has nothing to do with how it is implemented. The concept of a stack is the specification of how it is to be *accessed*. That is, the defining

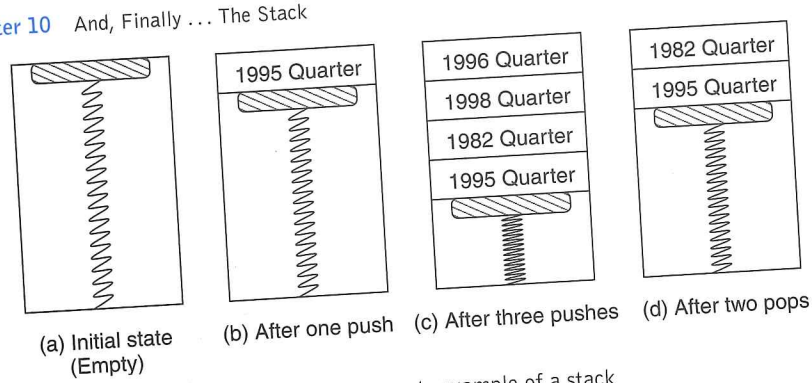


Figure 10.1 A coin holder in an auto armrest—example of a stack

ingredient of a stack is that the **last** thing you stored in it is the **first** thing you remove from it. That is what makes a stack different from everything else in the world. Simply put: Last In, First Out, or LIFO.

In the terminology of computer programming languages, we say the stack is an example of an *abstract data type*. That is, an abstract data type is a storage mechanism that is defined by the operations performed on it and not at all by the specific manner in which it is implemented. In Chapter 19, we will write programs in C that use linked lists, another example of an abstract data type.

10.1.2 Two Example Implementations

A coin holder in the armrest of an automobile is an example of a stack. The first quarter you take to pay the highway toll is the last quarter you added to the stack of quarters. As you add quarters, you push the earlier quarters down into the coin holder.

Figure 10.1 shows the behavior of a coin holder. Initially, as shown in Figure 10.1a, the coin holder is empty. The first highway toll is 75 cents, and you give the toll collector a dollar. She gives you 25 cents change, a 1995 quarter, which you insert into the coin holder. The coin holder appears as shown in Figure 10.1b.

There are special terms for the insertion and removal of elements from a stack. We say we *push* an element onto the stack when we insert it. We say we *pop* an element from the stack when we remove it.

The second highway toll is $4.25, and you give the toll collector $5.00. She gives you 75 cents change, which you insert into the coin holder: first a 1982 quarter, then a 1998 quarter, and finally, a 1996 quarter. Now the coin holder is as shown in Figure 10.1c. The third toll is 50 cents, and you remove (pop) the top two quarters from the coin holder: the 1996 quarter first and then the 1998 quarter. The coin holder is then as shown in Figure 10.1d.

The coin holder is an example of a stack, **precisely** because it obeys the LIFO requirement. Each time you insert a quarter, you do so at the top. Each time you remove a quarter, you do so from the top. The last coin you inserted is the first coin you remove; therefore, it is a stack.

Another implementation of a stack, sometimes referred to as a hardware stack, is shown in Figure 10.2. Its behavior resembles that of the coin holder we

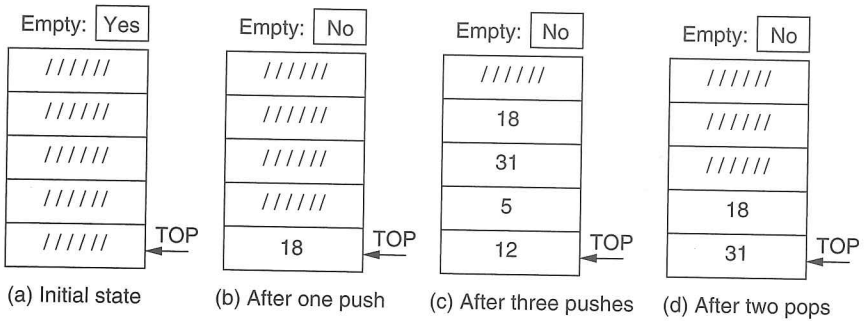


Figure 10.2 A stack, implemented in hardware—data entries move

just described. It consists of some number of registers, each of which can store an element. The example of Figure 10.2 contains five registers. As each element is added to the stack or removed from the stack, the elements **already** on the stack **move.**

In Figure 10.2a, the stack is initially shown as empty. Access is always via the first element, which is labeled TOP. If the value 18 is pushed on to the stack, we have Figure 10.2b. If the three values, 31, 5, and 12, are pushed (in that order), the result is Figure 10.2c. Finally, if two elements are popped from the stack, we have Figure 10.2d. The distinguishing feature of the stack of Figure 10.2 is that, like the quarters in the coin holder, as each value is added or removed, all the values already on the stack move.

10.1.3 Implementation in Memory

By far the most common implementation of a stack in a computer is as shown in Figure 10.3. The stack consists of a sequence of memory locations along with a mechanism, called the *stack pointer,* that keeps track of the top of the stack, that is, the location containing the most recent element pushed. Each value pushed is stored in one of the memory locations. In this case, the data already stored on the stack **does not physically move.**

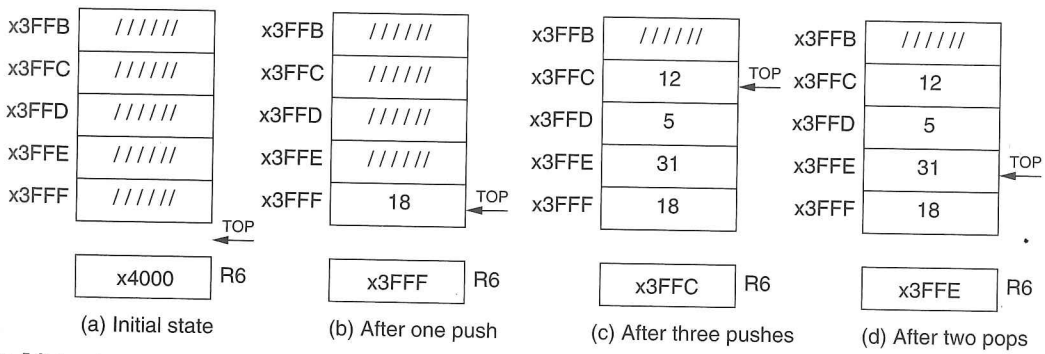


Figure 10.3 A stack, implemented in memory—data entries do not move

In the example shown in Figure 10.3, the stack consists of five locations, x3FFF through x3FFB. R6 is the stack pointer.

Figure 10.3a shows an initially empty stack. Figure 10.3b shows the stack after pushing the value 18. Figure 10.3c shows the stack after pushing the values 31, 5, and 12, in that order. Figure 10.3d shows the stack after popping the top two elements off the stack. Note that those top two elements (the values 5 and 12) are still present in memory locations x3FFD and x3FFC. However, as we will see momentarily, those values 5 and 12 cannot be accessed from memory, as long as the access to memory is controlled by the stack mechanism.

Push

In Figure 10.3a, R6 contains x4000, the address just ahead of the first (BASE) location in the stack. This indicates that the stack is initially empty. The BASE address of the stack of Figure 10.3 is x3FFF.

We first push the value 18 onto the stack, resulting in Figure 10.3b. The stack pointer provides the address of the last value pushed, in this case, x3FFF, where 18 is stored. Note that the contents of locations x3FFE, x3FFD, x3FFC, and x3FFB are not shown. As will be seen momentarily, the contents of these locations are irrelevant since they can never be accessed provided that locations x3FFF through x3FFB are accessed *only* as a stack.

When we push a value onto the stack, the stack pointer is decremented and the value stored. The two-instruction sequence

```
PUSH    ADD   R6,R6,#-1
        STR   R0,R6,#0
```

pushes the value contained in R0 onto the stack. Thus, for the stack to be as shown in Figure 10.3b, R0 must have contained the value 18 before the two-instruction sequence was executed.

The three values 31, 5, and 12 are pushed onto the stack by loading each in turn into R0, and then executing the two-instruction sequence. In Figure 10.3c, R6 (the stack pointer) contains x3FFC, indicating that 12 was the last element pushed.

Pop

To pop a value from the stack, the value is read and the stack pointer is incremented. The following two-instruction sequence

```
POP     LDR   R0,R6,#0
        ADD   R6,R6,#1
```

pops the value contained in the top of the stack and loads it into R0.

If the stack were as shown in Figure 10.3c and we executed the sequence twice, we would pop two values from the stack. In this case, we would first remove the 12, and then the 5. Assuming the purpose of popping two values is to use those two values, we would, of course, have to move the 12 from R0 to some other location before calling POP a second time.

Figure 10.3d shows the stack after that sequence of operations. R6 contains x3FFE, indicating that 31 is now at the top of the stack. Note that the values 12 and 5 are still stored in memory locations x3FFD and x3FFC, respectively. However, since the stack requires that we push by executing the PUSH sequence and pop by executing the POP sequence, we cannot access these two values if we obey the rules. The fancy name for "the rules" is the *stack protocol.*

Underflow

What happens if we now attempt to pop three values from the stack? Since only two values remain on the stack, we would have a problem. Attempting to pop items that have not been previously pushed results in an *underflow* situation. In our example, we can test for underflow by comparing the stack pointer with x4000, which would be the contents of R6 if there were nothing left on the stack to pop. If UNDERFLOW is the label of a routine that handles the underflow condition, our resulting POP sequence would be

```
POP     LD      R1,EMPTY
        ADD     R2,R6,R1        ; Compare stack
        BRz     UNDERFLOW       ; pointer with x4000.
        LDR     R0,R6,#0
        ADD     R6,R6,#1
        RET
EMPTY   .FILL   xC000           ; EMPTY <-- -x4000
```

Rather than have the POP routine immediately jump to the UNDERFLOW routine if the POP is unsuccessful, it is often useful to have the POP routine return to the calling program, with the underflow information contained in a register.

A common convention for doing this is to use a register to provide success/failure information. Figure 10.4 is a flowchart showing how the POP routine could be augmented, using R5 to report this success/failure information.

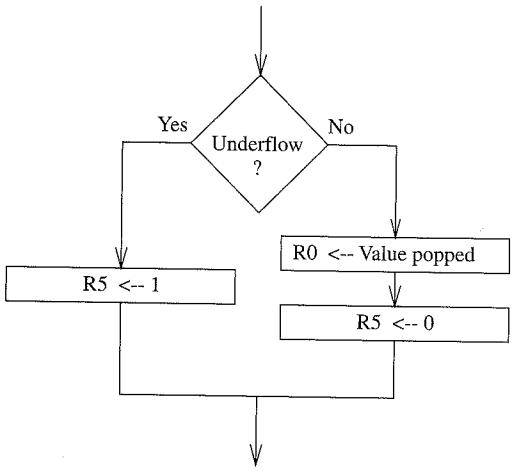


Figure 10.4 POP routine, including test for underflow

Upon return from the POP routine, the calling program would examine R5 to determine whether the POP completed successfully (R5 = 0), or not (R5 = 1).

Note that since the POP routine reports success or failure in R5, whatever was stored in R5 **before** the POP routine was called is lost. Thus, it is the job of the calling program to save the contents of R5 before the JSR instruction is executed. Recall from Section 9.1.7 that this is an example of a caller-save situation.

The resulting POP routine is shown in the following instruction sequence. Note that since the instruction immediately preceding the RET instruction sets/clears the condition codes, the calling program can simply test Z to determine whether the POP was completed successfully.

```
POP       LD      R1,EMPTY
          ADD     R2,R6,R1
          BRz     Failure
          LDR     R0,R6,#0
          ADD     R6,R6,#1
          AND     R5,R5,#0
          RET
Failure   AND     R5,R5,#0
          ADD     R5,R5,#1
          RET
EMPTY     .FILL   xC000          ; EMPTY <-- -x4000
```

Overflow

What happens when we run out of available space and we try to push a value onto the stack? Since we cannot store values where there is no room, we have an *overflow* situation. We can test for overflow by comparing the stack pointer with (in the example of Figure 10.3) x3FFB. If they are equal, we have no room to push another value onto the stack. If OVERFLOW is the label of a routine that handles the overflow condition, our resulting PUSH sequence would be

```
PUSH      LD      R1,MAX
          ADD     R2,R6,R1
          BRz     OVERFLOW
          ADD     R6,R6,#-1
          STR     R0,R6,#0
          RET
MAX       .FILL   xC005          ; MAX <-- -x3FFB
```

In the same way that it is useful to have the POP routine return to the calling program with success/failure information, rather than immediately jumping to the UNDERFLOW routine, it is useful to have the PUSH routine act similarly.

We augment the PUSH routine with instructions to store 0 (success) or 1 (failure) in R5, depending on whether or not the push completed successfully. Upon return from the PUSH routine, the calling program would examine R5 to determine whether the PUSH completed successfully (R5 = 0) or not (R5 = 1).

Note again that since the PUSH routine reports success or failure in R5, we have another example of a caller-save situation. That is, since whatever was stored in R5 before the PUSH routine was called is lost, it is the job of the calling program to save the contents of R5 before the JSR instruction is executed.

Also, note again that since the instruction immediately preceding the RET instruction sets/clears the condition codes, the calling program can simply test Z or P to determine whether the POP completed successfully (see the following PUSH routine).

```
PUSH        LD      R1,MAX
            ADD     R2,R6,R1
            BRz     Failure
            ADD     R6,R6,#-1
            STR     R0,R6,#0
            AND     R5,R5,#0
            RET
Failure     AND     R5,R5,#0
            ADD     R5,R5,#1
            RET
MAX         .FILL   xC005          ; MAX <-- -x3FFB
```

10.1.4 The Complete Picture

The POP and PUSH routines allow us to use memory locations x3FFF through x3FFB as a five-entry stack. If we wish to push a value onto the stack, we simply load that value into R0 and execute JSR PUSH. To pop a value from the stack into R0, we simply execute JSR POP. If we wish to change the location or the size of the stack, we adjust BASE and MAX accordingly.

Before leaving this topic, we should be careful to clean up one detail. The subroutines PUSH and POP make use of R1, R2, and R5. If we wish to use the values stored in those registers after returning from the PUSH or POP routine, we had best save them before using them. In the case of R1 and R2, it is easiest to save them in the PUSH and POP routines before using them and then to restore them before returning to the calling program. That way, the calling program does not even have to know that these registers are used in the PUSH and POP routines. This is an example of the callee-save situation described in Section 9.1.7. In the case of R5, the situation is different since the calling program does have to know the success or failure that is reported in R5. Thus, it is the job of the calling program to save the contents of R5 before the JSR instruction is executed if the calling program wishes to use the value stored there again. This is an example of the caller-save situation.

The final code for our PUSH and POP operations is shown in Figure 10.5.

```
01    ;
02    ; Subroutines for carrying out the PUSH and POP functions.  This
03    ; program works with a stack consisting of memory locations x3FFF
04    ; (BASE) through x3FFB (MAX).  R6 is the stack pointer.
05    ;
06    POP             ST      R2,Save2        ; are needed by POP.
07                    ST      R1,Save1
08                    LD      R1,BASE         ; BASE contains -x3FFF.
09                    ADD     R1,R1,#-1       ; R1 contains -x4000.
0A                    ADD     R2,R6,R1        ; Compare stack pointer to x4000.
0B                    BRz     fail_exit       ; Branch if stack is empty.
0C                    LDR     R0,R6,#0        ; The actual "pop"
0D                    ADD     R6,R6,#1        ; Adjust stack pointer.
0E                    BRnzp   success_exit
0F    PUSH            ST      R2,Save2        ; Save registers that
10                    ST      R1,Save1        ; are needed by PUSH.
11                    LD      R1,MAX          ; MAX contains -x3FFB
12                    ADD     R2,R6,R1        ; Compare stack pointer to -x3FFB.
13                    BRz     fail_exit       ; Branch if stack is full.
14                    ADD     R6,R6,#-1       ; Adjust stack pointer.
15                    STR     R0,R6,#0        ; The actual "push"
16    success_exit    LD      R1,Save1        ; Restore original
17                    LD      R2,Save2        ; register values.
18                    AND     R5,R5,#0        ; R5 <-- success.
19                    RET
1A    fail_exit       LD      R1,Save1        ; Restore original
1B                    LD      R2,Save2        ; register values.
1C                    AND     R5,R5,#0
1D                    ADD     R5,R5,#1        ; R5 <-- failure.
1E                    RET
1F    BASE            .FILL   xC001           ; BASE contains -x3FFF.
20    MAX             .FILL   xC005
21    Save1           .FILL   x0000
22    Save2           .FILL   x0000
```

Figure 10.5 The stack protocol

10.2 Interrupt-Driven I/O (Part 2)

Recall our discussion in Section 8.1.4 about interrupt-driven I/O as an alternative to polling. As you know, in polling, the processor wastes its time spinning its wheels, re-executing again and again the LDI and BR instructions until the Ready bit is set. With interrupt-driven I/O, none of that testing and branching has to go on. Instead, the processor spends its time doing what is hopefully useful work, executing some program, until it is notified that some I/O device needs attention.

You remember that there are two parts to interrupt-driven I/O:

1. the enabling mechanism that allows an I/O device to interrupt the processor when it has input to deliver or is ready to accept output, and
2. the process that manages the transfer of the I/O data.

In Section 8.5, we showed the enabling mechanism for interrupting the processor, that is, asserting the INT signal. We showed how the Ready bit, combined with the Interrupt Enable bit, provided an interrupt request signal. We showed that if the interrupt request signal is at a higher priority level (PL) than the PL of the currently executing process, the INT signal is asserted. We saw (Figure 8.8) that with this mechanism, the processor did not have to waste a lot of time polling. In Section 8.5, we could not study the process that manages the transfer of the I/O data because it involves the use of a stack, and you were not yet familiar with the stack. Now you know about stacks, so we can finish the explanation.

The actual management of the I/O data transfer goes through three stages, as shown in Figure 8.6:

1. Initiate the interrupt.
2. Service the interrupt.
3. Return from the interrupt.

We will discuss these in turn.

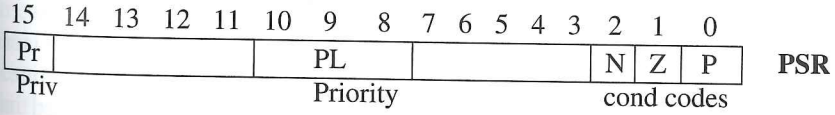
10.2.1 Initiate and Service the Interrupt

Recall from Section 8.5 (and Figure 8.8) that an interrupt is initiated because an I/O device with higher priority than the currently running program has caused the INT signal to be asserted. The processor, for its part, tests for the presence of INT each time it completes an instruction cycle. If the test is negative, business continues as usual and the next instruction of the currently running program is fetched. If the test is positive, that next instruction is not fetched.

Instead, preparation is made to interrupt the program that is running and execute the interrupt service routine that deals with the needs of the I/O device that has requested this higher priority service. Two steps must be carried out: (1) Enough of the state of the program that is running must be saved so we can later continue where we left off, and (2) enough of the state of the interrupt service routine must be loaded so we can begin to service the interrupt request.

The State of a Program

The state of a program is a snapshot of the contents of all the resources that the program affects. It includes the contents of the memory locations that are part of the program and the contents of all the general purpose registers. It also includes two very important registers, the PC and the PSR. The PC you are very familiar with; it contains the address of the next instruction to be executed. The PSR, shown here, is the Processor Status Register. It contains several important pieces of information about the status of the running program.



PSR[15] indicates whether the program is running in privileged (supervisor) or unprivileged (user) mode. In privileged mode, the program has access to

important resources not available to user programs. We will see momentarily why that is important in dealing with interrupts. PSR[10:8] specifies the priority level (PL) or sense of urgency of the execution of the program. As has been mentioned previously, there are eight priority levels, PL0 (lowest) to PL7 (highest). Finally, PSR[2:0] is used to store the condition codes. PSR[2] is the N bit, PSR[1] is the Z bit, and PSR[0] is the P bit.

Saving the State of the Interrupted Program

The first step in initiating the interrupt is to save enough of the state of the program that is running so it can continue where it left off after the I/O device request has been satisfied. That means, in the case of the LC-3, saving the PC and the PSR. The PC must be saved since it knows which instruction should be executed next when the interrupted program resumes execution. The condition codes (the N, Z, and P flags) must be saved since they may be needed by a subsequent conditional branch instruction after the program resumes execution. The priority level of the interrupted program must be saved because it specifies the urgency of the interrupted program with respect to all other programs. When the interrupted program resumes execution, it is important to know what priority level programs can interrupt it again and which ones can not. Finally, the privilege level of the program must be saved since it contains information about what processor resources the interrupted program can and can not access.

It is not necessary to save the contents of the general purpose registers since we assume that the service routine will save the contents of any general purpose register it needs before using it, and will restore it before returning to the interrupted program.

The LC-3 saves this state information on a special stack, called the Supervisor Stack, that is used only by programs that execute in privileged mode. A section of memory is dedicated for this purpose. This stack is separate from the User Stack, which is accessed by user programs. Programs access both stacks using R6 as the stack pointer. When accessing the Supervisor Stack, R6 is the Supervisor Stack Pointer. When accessing the User Stack, R6 is the User Stack Pointer. Two internal registers, Saved.SSP and Saved.USP, are used to save the stack pointer not in use. When the privilege mode changes from user to supervisor, the contents of R6 are saved in Saved.USP, and R6 is loaded with the contents of Saved.SSP before processing begins.

That is, before the interrupt service routine starts, R6 is loaded with the contents of the Supervisor Stack Pointer. Then PC and PSR of the interrupted program are pushed onto the Supervisor Stack, where they remain unmolested while the service routine executes.

Loading the State of the Interrupt Service Routine

Once the state of the interrupted program has been safely saved on the Supervisor Stack, the second step is to load the PC and PSR of the interrupt service routine. Interrupt service routines are similar to the trap service routines discussed in Chapter 9. They are program fragments stored in some prearranged set of locations in memory. They service interrupt requests.

Most processors use the mechanism of *vectored interrupts*. You are familiar with this notion from your study of the trap vector contained in the TRAP instruction. In the case of interrupts, the 8-bit vector is provided by the device that is requesting the processor be interrupted. That is, the I/O device transmits to the processor an 8-bit interrupt vector along with its interrupt request signal and its priority level. The interrupt vector corresponding to the highest priority interrupt request is the one supplied to the processor. It is designated INTV. If the interrupt is taken, the processor expands the 8-bit interrupt vector (INTV) to form a 16-bit address, which is an entry into the Interrupt Vector Table. Recall from Chapter 9 that the Trap Vector Table consists of memory locations x0000 to x00FF, each containing the starting address of a trap service routine. The Interrupt Vector Table consists of memory locations x0100 to x01FF, each containing the starting address of an interrupt service routine. The processor loads the PC with the contents of the address formed by expanding the interrupt vector INTV.

The PSR is loaded as follows: Since no instructions in the service routine have yet executed, PSR[2:0] is initially loaded with zeros. Since the interrupt service routine runs in privileged mode, PSR[15] is set to 0. PSR[10:8] is set to the priority level associated with the interrupt request.

This completes the initiation phase and the interrupt service routine is ready to go.

Service the Interrupt

Since the PC contains the starting address of the interrupt service routine, the service routine will execute, and the requirements of the I/O device will be serviced.

For example, the LC-3 keyboard could interrupt the processor every time a key is pressed by someone sitting at the keyboard. The keyboard interrupt vector would indicate the handler to invoke. The handler would then copy the contents of the data register into some preestablished location in memory.

10.2.2 Return from the Interrupt

The last instruction in every interrupt service routine is RTI, return from interrupt. When the processor finally accesses the RTI instruction, all the requirements of the I/O device have been taken care of.

Execution of the **RTI** instruction (opcode = 1000) consists simply of popping the PSR and the PC from the Supervisor Stack (where they have been resting peacefully) and restoring them to their rightful places in the processor. The condition codes are now restored to what they were when the program was interrupted, in case they are needed by a subsequent BR instruction in the program. PSR[15] and PSR[10:8] now reflect the privilege level and priority level of the about-to-be-resumed program. Similarly, the PC is restored to the address of the instruction that would have been executed next if the program had not been interrupted.

With all these things as they were before the interrupt occurred, the program can resume as if nothing had happened.

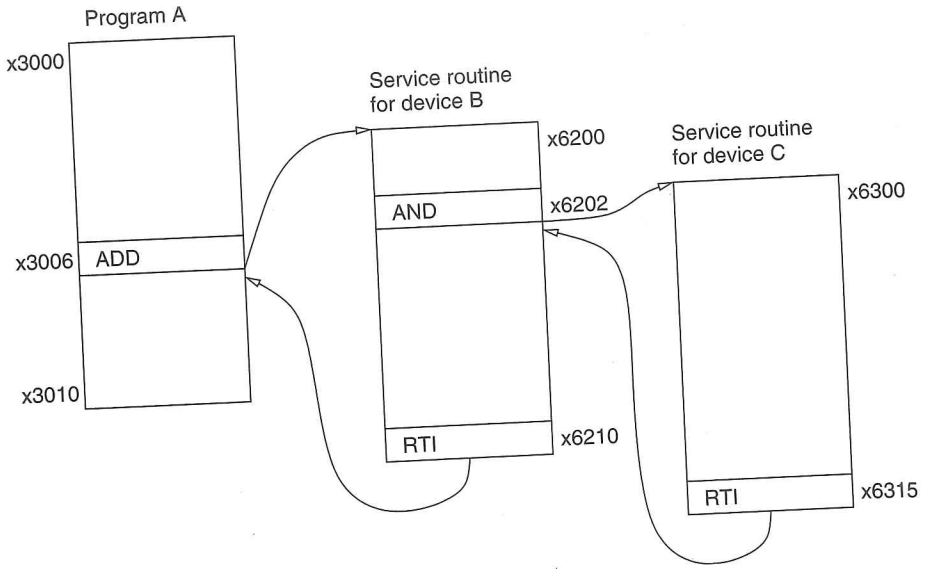


Figure 10.6 Execution flow for interrupt-driven I/O

10.2.3 An Example

We complete the discussion of interrupt-driven I/O with an example.

Suppose program A is executing when I/O device B, having a PL higher than that of A, requests service. During the execution of the service routine for I/O device B, a still more urgent device C requests service.

Figure 10.6 shows the execution flow that must take place.

Program A consists of instructions in locations x3000 to x3010 and was in the middle of executing the ADD instruction at x3006, when device B sent its interrupt request signal and accompanying interrupt vector xF1, causing INT to be asserted.

Note that the interrupt service routine for device B is stored in locations x6200 to x6210; x6210 contains the RTI instruction. Note that the service routine for B was in the middle of executing the AND instruction at x6202, when device C sent its interrupt request signal and accompanying interrupt vector xF2. Since the request associated with device C is of a higher priority than that of device B, INT is again asserted.

Note that the interrupt service routine for device C is stored in locations x6300 to x6315; x6315 contains the RTI instruction.

Let us examine the order of execution by the processor. Figure 10.7 shows several snapshots of the contents of the Supervisor Stack and the PC during the execution of this example.

The processor executes as follows: Figure 10.7a shows the Supervisor Stack and the PC before program A fetches the instruction at x3006. Note that the stack pointer is shown as Saved.SSP, not R6. Since the interrupt has not yet occurred, R6 is pointing to the current contents of the User Stack. The INT signal (caused by an interrupt from device B) is detected at the end of execution of the instruction

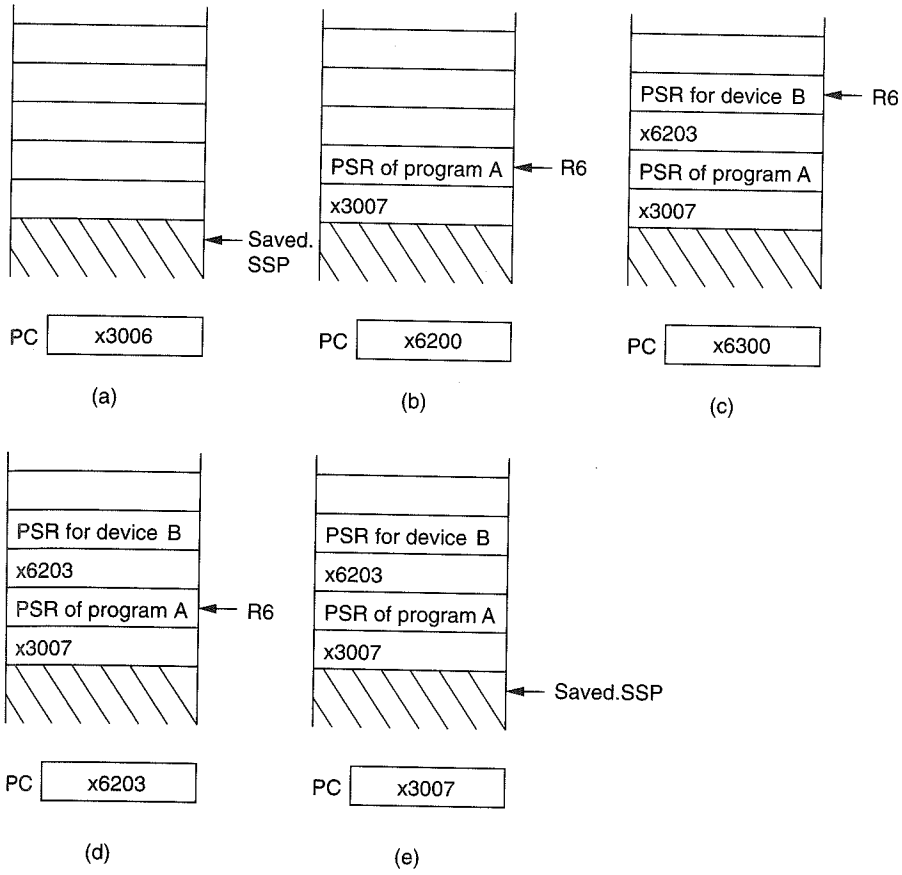


Figure 10.7 Snapshots of the contents of the Supervisor Stack and the PC during interrupt-driven I/O

in x3006. Since the state of program A must be saved on the Supervisor Stack, the first step is to start using the Supervisor Stack. This is done by saving R6 in the Saved.USP register, and loading R6 with the contents of the Saved.SSP register. The address x3007, the PC for the next instruction to be executed in program A, is pushed onto the stack. The PSR of program A, which includes the condition codes produced by the ADD instruction, is pushed onto the stack. The interrupt vector associated with device B is expanded to 16 bits x01F1, and the contents of x01F1 (x6200) are loaded into the PC. Figure 10.7b shows the stack and PC at this point.

The service routine for device B executes until a higher priority interrupt is detected at the end of execution of the instruction at x6202. The address x6203 is pushed onto the stack, along with the PSR of the service routine for B, which includes the condition codes produced by the AND instruction. The interrupt vector associated with device C is expanded to 16 bits (x01F2), and the contents of x01F2 (x6300) are loaded into the PC. Figure 10.7c shows the Supervisor Stack and PC at this point.

The interrupt service routine for device C executes to completion, finishing with the RTI instruction in x6315. The Supervisor Stack is popped twice, restoring the PSR of the service routine for device B, including the condition codes produced by the AND instruction in x6202, and restoring the PC to x6203. Figure 10.7d shows the stack and PC at this point.

The interrupt service routine for device B resumes execution at x6203 and runs to completion, finishing with the RTI instruction in x6210. The Supervisor Stack is popped twice, restoring the PSR of program A, including the condition codes produced by the ADD instruction in x3006, and restoring the PC to x3007. Finally, since program A is in User Mode, the contents of R6 are stored in Saved.SSP and R6 is loaded with the contents of Saved.USP. Figure 10.7e shows the Supervisor Stack and PC at this point.

Program A resumes execution with the instruction at x3007.

10.3 Arithmetic Using a Stack

10.3.1 The Stack as Temporary Storage

There are computers that use a stack instead of general purpose registers to store temporary values during a computation. Recall that our ADD instruction

```
ADD     R0,R1,R2
```

takes source operands from R1 and R2 and writes the result of the addition into R0. We call the LC-3 a *three-address machine* because all three locations (the two sources and the destination) are explicitly identified. Some computers use a stack for source and destination operands and explicitly identify **none** of them. The instruction would simply be

```
ADD
```

We call such a computer a stack machine, or a *zero-address machine*. The hardware would know that the source operands are the top two elements on the stack, which would be popped and then supplied to the ALU, and that the result of the addition would be pushed onto the stack.

To perform an ADD on a stack machine, the hardware would execute two pops, an add, and a push. The two pops would remove the two source operands from the stack, the add would compute their sum, and the push would place the result back on the stack. Note that the pop, push, and add are not part of the ISA of that computer, and therefore not available to the programmer. They are control signals that the hardware uses to make the actual pop, push, and add occur. The control signals are part of the microarchitecture, similar to the load enable signals and mux select signals we discussed in Chapters 4 and 5. As is the case with LC-3 instructions LD and ST, and control signals PCMUX and LD.MDR, the programmer simply instructs the computer to ADD, and the microarchitecture does the rest.

Sometimes (as we will see in our final example of this chapter), it is useful to process arithmetic using a stack. Intermediate values are maintained on the

stack rather than in general purpose registers, such as the LC-3's R0 through R7. Most general purpose microprocessors, including the LC-3, use general purpose registers. Most calculators use a stack.

10.3.2 An Example

For example, suppose we wanted to evaluate $(A + B) \cdot (C + D)$, where A contains 25, B contains 17, C contains 3, and D contains 2, and store the result in E. If the LC-3 had a multiply instruction (we would probably call it MUL), we could use the following program:

```
LD    R0,A
LD    R1,B
ADD   R0,R0,R1
LD    R2,C
LD    R3,D
ADD   R2,R2,R3
MUL   R0,R0,R2
ST    R0,E
```

With a calculator, we could execute the following eight operations:

```
(1)    push     25
(2)    push     17
(3)    add
(4)    push     3
(5)    push     2
(6)    add
(7)    multiply
(8)    pop      E
```

with the final result popped being the result of the computation, that is, 210. Figure 10.8 shows a snapshot of the stack after each of the eight operations.

In Section 10.5, we write a program to cause the LC-3 (with keyboard and monitor) to act like such a calculator. We say the LC-3 *simulates* the calculator when it executes that program.

But first, let's examine the subroutines we need to conduct the various arithmetic operations.

10.3.3 OpAdd, OpMult, and OpNeg

The calculator we simulate in Section 10.5 has the ability to enter values, add, subtract, multiply, and display results. To add, subtract, and multiply, we need three subroutines:

1. OpAdd, which will pop two values from the stack, add them, and push the result onto the stack.

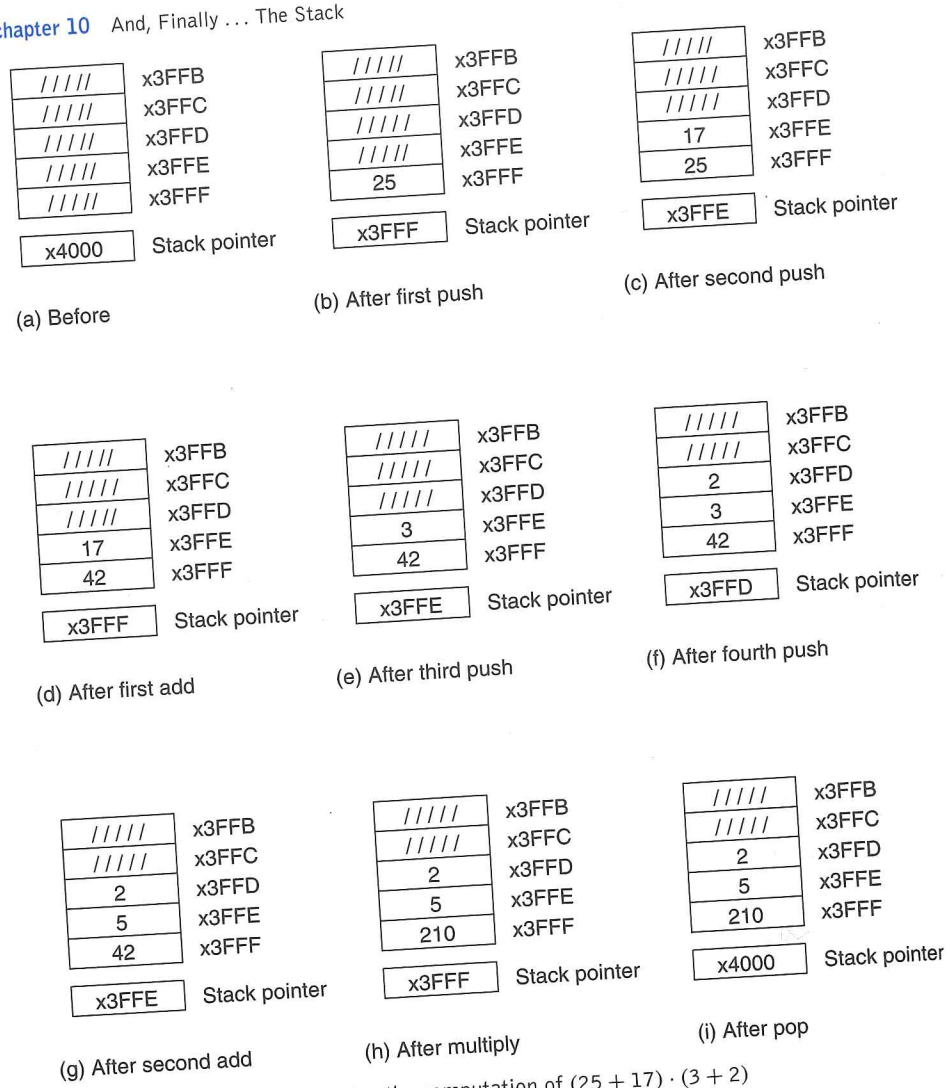


Figure 10.8 Stack usage during the computation of $(25 + 17) \cdot (3 + 2)$

2. OpMult, which will pop two values from the stack, multiply them, and push the result onto the stack.
3. OpNeg, which will pop the top value, form its 2's complement negative value, and push the result onto the stack.

The OpAdd Algorithm

Figure 10.9 shows the flowchart of the OpAdd algorithm. Basically, the algorithm attempts to pop two values off the stack and, if successful, add them. If the result is within the range of acceptable values (that is, an integer between -999 and $+999$), then the result is pushed onto the stack.

There are two things that could prevent the OpAdd algorithm from completing successfully: Fewer than two values are available on the stack for source operands,

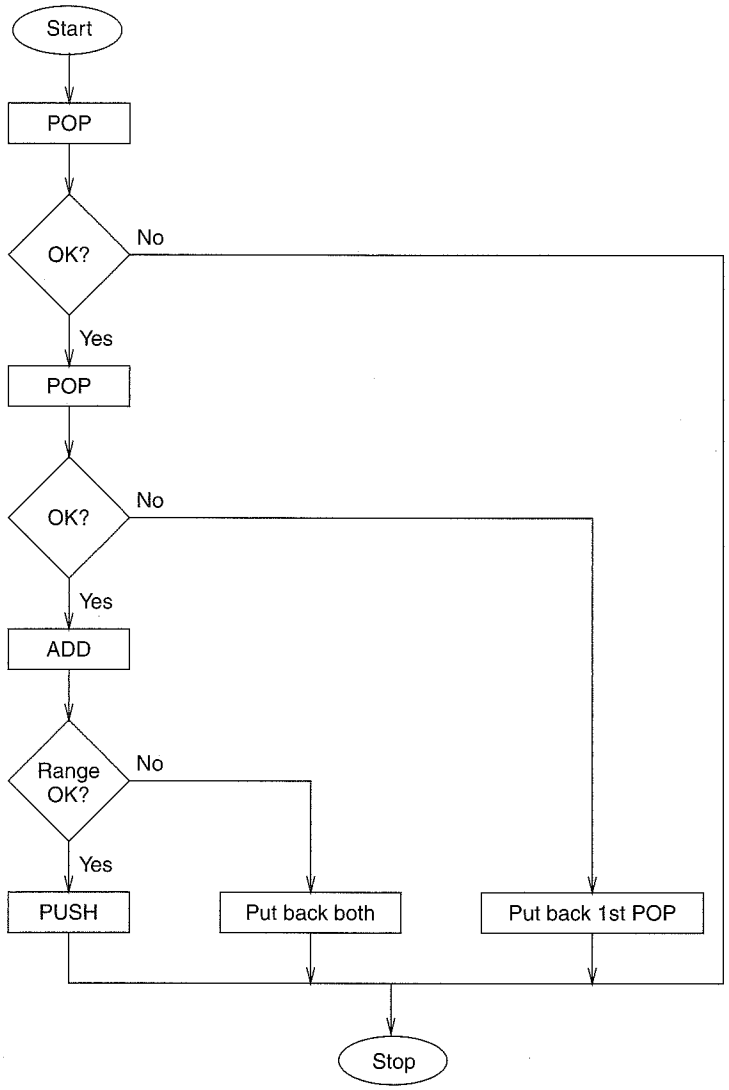


Figure 10.9 Flowchart for OpAdd algorithm

or the result is out of range. In both cases, the stack is put back to the way it was at the start of the OpAdd algorithm, a 1 is stored in R5 to indicate failure, and control is returned to the calling program. If the first pop is unsuccessful, the stack is not changed since the POP routine leaves the stack as it was. If the second of the two pops reports back unsuccessfully, the stack pointer is decremented, which effectively returns the first value popped to the top of the stack. If the result is outside the range of acceptable values, then the stack pointer is decremented twice, returning both values to the top of the stack.

The OpAdd algorithm is shown in Figure 10.10.

Note that the OpAdd algorithm calls the RangeCheck algorithm. This is a simple test to be sure the result of the computation is within what can successfully

```
01    ;
02    ;       Routine to pop the top two elements from the stack,
03    ;       add them, and push the sum onto the stack.  R6 is
04    ;       the stack pointer.
05    ;
06    OpAdd   JSR     POP             ; Get first source operand.
07            ADD     R5,R5,#0        ; Test if POP was successful.
08            BRp     Exit            ; Branch if not successful.
09            ADD     R1,R0,#0        ; Make room for second operand.
0A            JSR     POP             ; Get second source operand.
0B            ADD     R5,R5,#0        ; Test if POP was successful.
0C            BRp     Restore1        ; Not successful, put back first.
0D            ADD     R0,R0,R1        ; THE Add.
0E            JSR     RangeCheck      ; Check size of result.
0F            BRp     Restore2        ; Out of range, restore both.
10            JSR     PUSH            ; Push sum on the stack.
11            RET                     ; On to the next task...
12    Restore2  ADD   R6,R6,#-1       ; Decrement stack pointer.
13    Restore1  ADD   R6,R6,#-1       ; Decrement stack pointer.
14    Exit      RET
```

Figure 10.10 The OpAdd algorithm

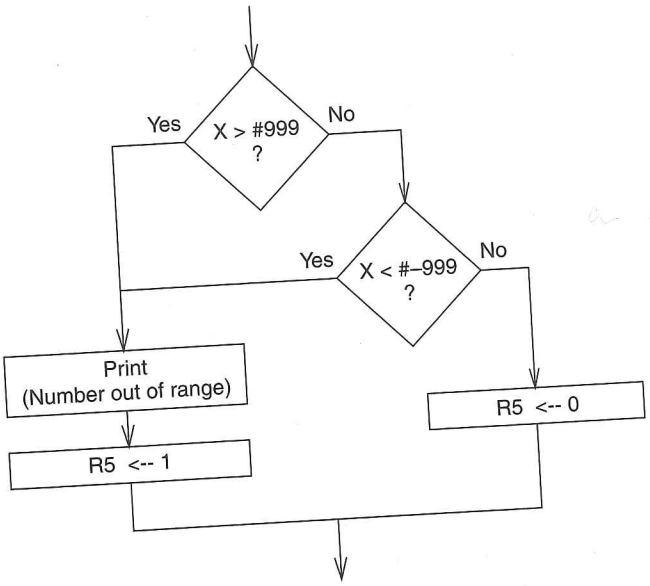


Figure 10.11 The RangeCheck algorithm flowchart

be stored in a single stack location. For our purposes, suppose we restrict values to integers in the range −999 to +999. This will come in handy in Section 10.5 when we design our home-brew calculator. The flowchart for the RangeCheck algorithm is shown in Figure 10.11. The LC-3 program that implements this algorithm is shown in Figure 10.12.

```
01   ;
02   ;     Routine to check that the magnitude of a value is
03   ;     between -999 and +999.
04   ;
05   RangeCheck    LD        R5,Neg999
06                 ADD       R4,R0,R5    ; Recall that R0 contains the
07                 BRp       BadRange    ; result being checked.
08                 LD        R5,Pos999
09                 ADD       R4,R0,R5
0A                 BRn       BadRange
0B                 AND       R5,R5,#0    ; R5 <-- success
0C                 RET
0D   BadRange      ST        R7,Save     ; R7 is needed by TRAP/RET.
0E                 LEA       R0,RangeErrorMsg
0F                 TRAP      x22         ; Output character string
10                 LD        R7,Save
11                 AND       R5,R5,#0    ;
12                 ADD       R5,R5,#1    ; R5 <-- failure
13                 RET
14   Neg999        .FILL     #-999
15   Pos999        .FILL     #999
16   Save          .FILL     x0000
17   RangeErrorMsg .FILL     x000A
18                 .STRINGZ  "Error: Number is out of range."
```

Figure 10.12 The RangeCheck algorithm

The OpMult Algorithm

Figure 10.13 shows the flowchart of the OpMult algorithm, and Figure 10.14 shows the LC-3 program that implements that algorithm. Similar to the OpAdd algorithm, the OpMult algorithm attempts to pop two values off the stack and, if successful, multiplies them. Since the LC-3 does not have a multiply instruction, multiplication is performed as we have done in the past as a sequence of adds. Lines 17 to 19 of Figure 10.14 contain the crux of the actual multiply. If the result is within the range of acceptable values, then the result is pushed onto the stack.

If the second of the two pops reports back unsuccessfully, the stack pointer is decremented, which effectively returns the first value popped to the top of the stack. If the result is outside the range of acceptable values, which as before will be indicated by a 1 in R5, then the stack pointer is decremented twice, returning both values to the top of the stack.

The OpNeg Algorithm

We have provided algorithms to add and multiply the top two elements on the stack. To subtract the top two elements on the stack, we can use our OpAdd algorithm if we first replace the top of the stack with its negative value. That is, if the top of the stack contains A, and the second element on the stack contains B,

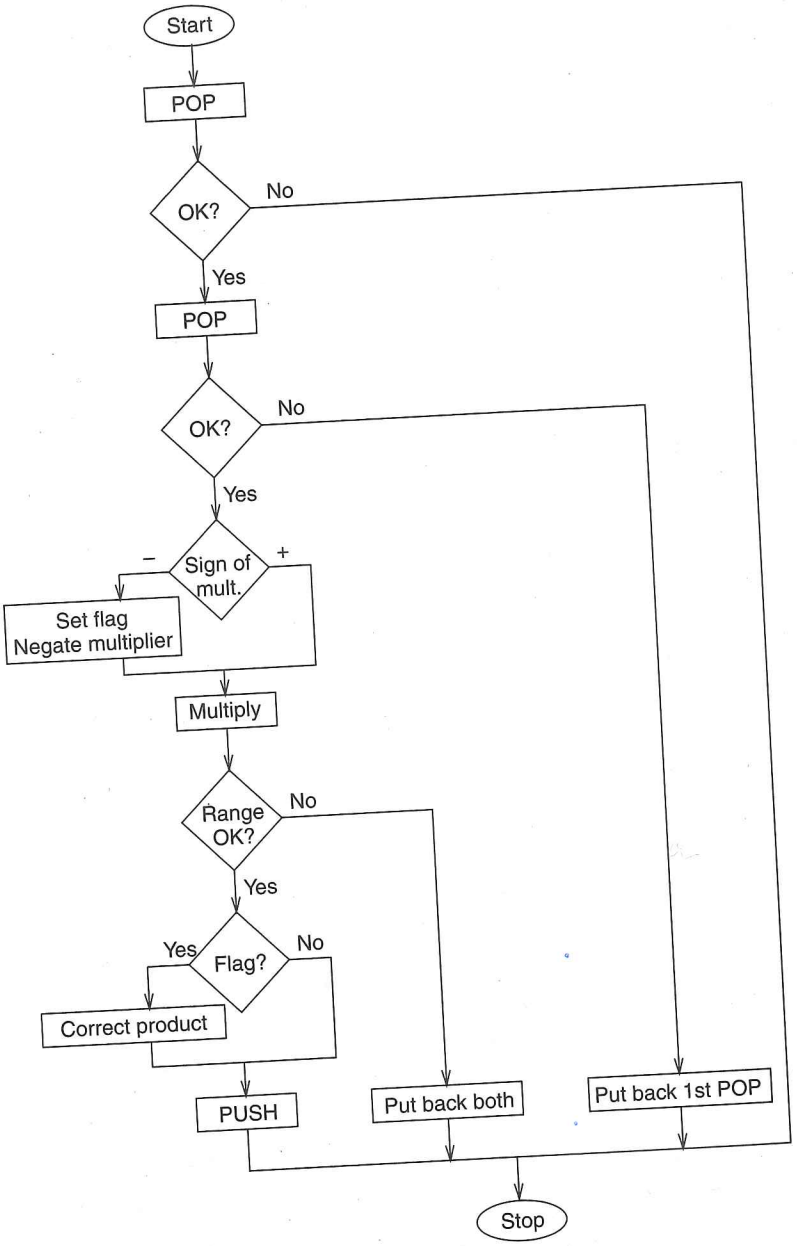


Figure 10.13 Flowchart for the OpMult algorithm

```
01  ;
02  ;     Algorithm to pop two values from the stack, multiply them,
03  ;     and if their product is within the acceptable range, push
04  ;     the result onto the stack.  R6 is stack pointer.
05  ;
06  OpMult          AND    R3,R3,#0       ; R3 holds sign of multiplier.
07                  JSR    POP            ; Get first source from stack.
08                  ADD    R5,R5,#0       ; Test for successful POP.
09                  BRp    Exit           ; Failure
0A                  ADD    R1,R0,#0       ; Make room for next POP.
0B                  JSR    POP            ; Get second source operand.
0C                  ADD    R5,R5,#0       ; Test for successful POP.
0D                  BRp    Restore1       ; Failure; restore first POP.
0E                  ADD    R2,R0,#0       ; Moves multiplier, tests sign.
0F                  BRzp   PosMultiplier
10                  ADD    R3,R3,#1       ; Sets FLAG: Multiplier is neg.
11                  NOT    R2,R2
12                  ADD    R2,R2,#1       ; R2 contains -(multiplier).
13  PosMultiplier   AND    R0,R0,#0       ; Clear product register.
14                  ADD    R2,R2,#0
15                  BRz    PushMult       ; Multiplier = 0, Done.
16  ;
17  MultLoop        ADD    R0,R0,R1       ; THE actual "multiply"
18                  ADD    R2,R2,#-1      ; Iteration Control
19                  BRp    MultLoop
1A  ;
1B                  JSR    RangeCheck
1C                  ADD    R5,R5,#0       ; R5 contains success/failure.
1D                  BRp    Restore2
1E  ;
1F                  ADD    R3,R3,#0       ; Test for negative multiplier.
20                  BRz    PushMult
21                  NOT    R0,R0          ; Adjust for
22                  ADD    R0,R0,#1       ; sign of result.
23  PushMult        JSR    PUSH           ; Push product on the stack.
24                  RET
25  Restore2        ADD    R6,R6,#-1      ; Adjust stack pointer.
26  Restore1        ADD    R6,R6,#-1      ; Adjust stack pointer.
27  Exit            RET
```

Figure 10.14 The OpMult algorithm

and we wish to pop A, B and push B−A, we can accomplish this by first negating the top of the stack and then performing OpAdd.

The algorithm for negating the element on the top of the stack, OpNeg, is shown in Figure 10.15.

```
01    ; Algorithm to pop the top of the stack, form its negative,
02    ; and push the result onto the stack.
03    ;
04    OpNeg   JSR     POP         ; Get the source operand.
05            ADD     R5,R5,#0    ; Test for successful pop
06            BRp     Exit        ; Branch if failure.
07            NOT     R0,R0
08            ADD     R0,R0,#1    ; Form the negative of source.
09            JSR     PUSH        ; Push result onto the stack.
0A    Exit    RET
```

Figure 10.15 The OpNeg algorithm

10.4 Data Type Conversion

It has been a long time since we talked about data types. We have been exposed to several data types: unsigned integers for address arithmetic, 2's complement integers for integer arithmetic, 16-bit binary strings for logical operations, floating point numbers for scientific computation, and ASCII codes for interaction with input and output devices.

It is important that every instruction be provided with source operands of the data type that the instruction requires. For example, **ADD** requires operands that are 2's complement integers. If the ALU were supplied with floating point operands, the computer would produce garbage results.

It is not uncommon in high-level language programs to find an instruction of the form $A = R + I$ where R (floating point) and I (2's complement integer) are represented in different data types.

If the operation is to be performed by a floating point adder, then we have a problem with I. To handle the problem, one must first convert the value I from its original data type (2's complement integer) to the data type required by the operation (floating point).

Even the LC-3 has this data type conversion problem. Consider a multiple-digit integer that has been entered via the keyboard. It is represented as a string of ASCII characters. To perform arithmetic on it, you must first convert the value to a 2's complement integer. Consider a 2's complement representation of a value that you wish to display on the monitor. To do so, you must first convert it to an ASCII string.

In this section, we will examine routines to convert between ASCII strings of decimal digits and 2's complement binary integers.

10.4.1 Example: The Bogus Program: 2 + 3 = e

First, let's examine Figure 10.16, a concrete example of how one can get into trouble if one is not careful about keeping track of the data type of each of the values with which one is working.

Suppose we wish to enter two digits from the keyboard, add them, and display the results on the monitor. At first blush, we write the simple program of Figure 10.16. What happens?

```
01    TRAP    x23          ; Input from the keyboard.
02    ADD     R1,R0,#0     ; Make room for another input.
03    TRAP    x23          ; Input another character.
04    ADD     R0,R1,R0     ; Add the two inputs.
05    TRAP    x21          ; Display result on the monitor.
06    TRAP    x25          ; Halt.
```

Figure 10.16 ADDITION without paying attention to data types

Suppose the first digit entered via the keyboard is a 2 and the second digit entered via the keyboard is a 3. What will be displayed on the monitor before the program terminates? The value loaded into R0 as a result of entering a 2 is the ASCII code for 2, which is x0032. When the 3 is entered, the ASCII code for 3, which is x0033, will be loaded. Thus, the ADD instruction will add the two binary strings x0032 and x0033, producing x0065. When that value is displayed on the monitor, it will be treated as an ASCII code. Since x0065 is the ASCII code for a lowercase *e*, that is what will be displayed on the monitor.

The reason why we did not get 5 (which, at last calculation, was the correct result when adding 2 + 3) was that we didn't (*a*) convert the two input characters from ASCII to 2's complement integers before performing addition and (*b*) convert the result back to ASCII before displaying it on the monitor.

Exercise: Correct Figure 10.16 so that it will add two single-digit positive integers and give a single-digit positive sum. Assume that the two digits being added do in fact produce a single-digit sum.

10.4.2 ASCII to Binary

It is often useful to deal with numbers that require more than one digit to express them. Figure 10.17 shows the ASCII representation of the three-digit number 295, stored as an ASCII string in three consecutive LC-3 memory locations, starting at ASCIIBUFF.R1 contains the number of decimal digits in the number.

Note that in Figure 10.17, a whole LC-3 word (16 bits) is allocated for each ASCII character. One can (and, in fact, more typically, one does) store each ASCII character in a single byte of memory. In this example, we have decided to give each ASCII character its own word of memory in order to simplify the algorithm.

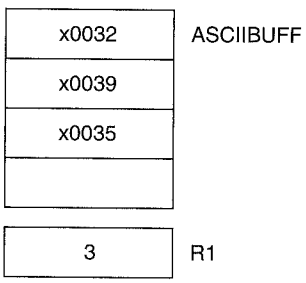


Figure 10.17 The ASCII representation of 295 stored in consecutive memory locations

Figure 10.18 shows the flowchart for converting the ASCII representation of Figure 10.17 into a binary integer. The value represented must be in the range 0 to +999, that is, it is limited to three decimal digits.

The algorithm systematically takes each digit, converts it from its ASCII code to its binary code by stripping away all but the last four bits, and then uses it to index into a table of 10 binary values, each corresponding to the value of one

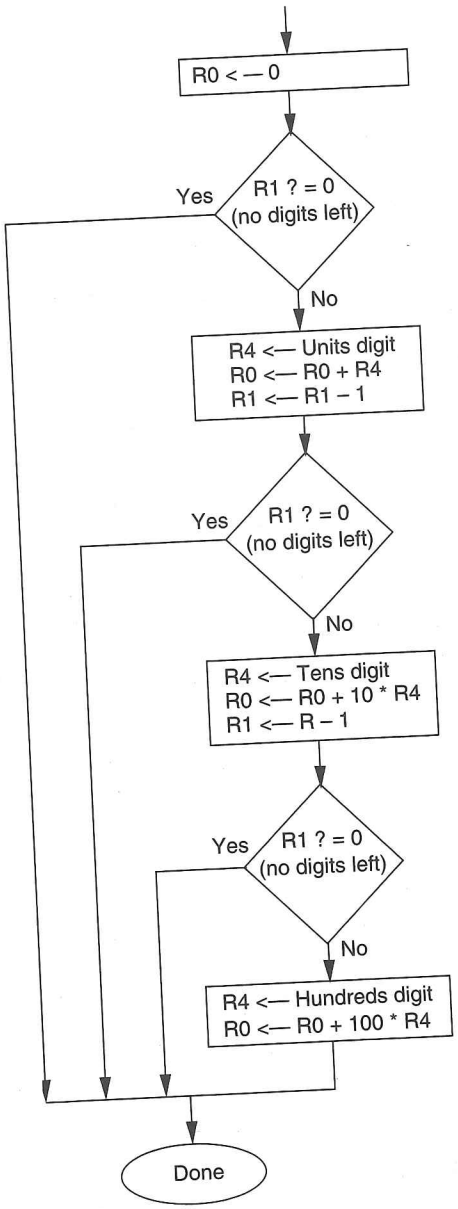


Figure 10.18 Flowchart, algorithm for ASCII-to-binary conversion

of the 10 digits. That value is then added to R0. R0 is used to accumulate the contributions of all the digits. The result is returned in R0.

Figure 10.19 shows the LC-3 program that implements this algorithm.

```
01   ;
02   ;  This algorithm takes an ASCII string of three decimal digits and
03   ;  converts it into a binary number.  R0 is used to collect the result.
04   ;  R1 keeps track of how many digits are left to process.  ASCIIBUFF
05   ;  contains the most significant digit in the ASCII string.
06   ;
07   ASCIItoBinary  AND    R0,R0,#0        ; R0 will be used for our result.
08                  ADD    R1,R1,#0        ; Test number of digits.
09                  BRz    DoneAtoB        ; There are no digits.
0A   ;
0B                  LD     R3,NegASCIIOffset  ; R3 gets xFFD0, i.e., -x0030.
0C                  LEA    R2,ASCIIBUFF
0D                  ADD    R2,R2,R1
0E                  ADD    R2,R2,#-1       ; R2 now points to "ones" digit.
0F   ;
10                  LDR    R4,R2,#0        ; R4 <-- "ones" digit
11                  ADD    R4,R4,R3        ; Strip off the ASCII template.
12                  ADD    R0,R0,R4        ; Add ones contribution.
13   ;
14                  ADD    R1,R1,#-1
15                  BRz    DoneAtoB        ; The original number had one digit.
16                  ADD    R2,R2,#-1       ; R2  now points to "tens" digit.
17   ;
18                  LDR    R4,R2,#0        ; R4 <-- "tens" digit
19                  ADD    R4,R4,R3        ; Strip off ASCII template.
1A                  LEA    R5,LookUp10     ; LookUp10 is BASE of tens values.
1B                  ADD    R5,R5,R4        ; R5 points to the right tens value.
1C                  LDR    R4,R5,#0
1D                  ADD    R0,R0,R4        ; Add tens contribution to total.
1E   ;
1F                  ADD    R1,R1,#-1
20                  BRz    DoneAtoB        ; The original number had two digits.
21                  ADD    R2,R2,#-1       ; R2 now points to "hundreds" digit.
22   ;
23                  LDR    R4,R2,#0        ; R4 <-- "hundreds" digit
24                  ADD    R4,R4,R3        ; Strip off ASCII template.
25                  LEA    R5,LookUp100    ; LookUp100 is hundreds BASE.
26                  ADD    R5,R5,R4        ; R5 points to hundreds value.
27                  LDR    R4,R5,#0
28                  ADD    R0,R0,R4        ; Add hundreds contribution to total.
29   ;
2A   DoneAtoB       RET
2B   NegASCIIOffset .FILL  xFFD0
2C   ASCIIBUFF      .BLKW  4
2D   LookUp10       .FILL  #0
```

Figure 10.19 ASCII-to-binary conversion routine

```
2E                      .FILL   #10
2F                      .FILL   #20
30                      .FILL   #30
31                      .FILL   #40
32                      .FILL   #50
33                      .FILL   #60
34                      .FILL   #70
35                      .FILL   #80
36                      .FILL   #90
37      ;
38      LookUp100       .FILL   #0
39                      .FILL   #100
3A                      .FILL   #200
3B                      .FILL   #300
3C                      .FILL   #400
3D                      .FILL   #500
3E                      .FILL   #600
3F                      .FILL   #700
40                      .FILL   #800
41                      .FILL   #900
```

Figure 10.19 ASCII-to-binary conversion routine (continued)

10.4.3 Binary to ASCII

Similarly, it is useful to convert the 2's complement integer into an ASCII string so that it can be displayed on the monitor. Figure 10.20 shows the algorithm for converting a 2's complement integer stored in R0 into an ASCII string stored in four consecutive memory locations, starting at ASCIIBUFF. The value initially in R0 is restricted to be within the range −999 to +999. After the algorithm completes execution, ASCIIBUFF contains the sign of the value initially stored in R0. The following three locations contain the three ASCII codes corresponding to the three decimal digits representing its magnitude.

The algorithm works as follows: First, the sign of the value is determined, and the appropriate ASCII code is stored. The value in R0 is replaced by its absolute value. The algorithm determines the hundreds-place digit by repeatedly subtracting 100 from R0 until the result goes negative. This is next repeated for the tens-place digit. The value left is the ones digit.

Exercise: [Very challenging] Suppose the decimal number is arbitrarily long. Rather than store a table of 10 values for the thousands-place digit, another table for the 10 ten-thousands-place digit, and so on, design an algorithm to do the conversion without resorting to any tables whatsoever. See Exercise 10.20.

Exercise: This algorithm always produces a string of four characters independent of the sign and magnitude of the integer being converted. Devise an algorithm that eliminates unnecessary characters in common representations, that is, an algorithm that does not store leading zeros nor a leading + sign. See Exercise 10.22.

```
01   ;
02   ;  This algorithm takes the 2's complement representation of a signed
03   ;  integer within the range -999 to +999 and converts it into an ASCII
04   ;  string consisting of a sign digit, followed by three decimal digits.
05   ;  R0 contains the initial value being converted.
06   ;
07   BinarytoASCII  LEA   R1,ASCIIBUFF  ; R1 points to string being generated.
08                  ADD   R0,R0,#0      ; R0 contains the binary value.
09                  BRn   NegSign       ;
0A                  LD    R2,ASCIIplus  ; First store the ASCII plus sign.
0B                  STR   R2,R1,#0
0C                  BRnzp Begin100
0D   NegSign        LD    R2,ASCIIminus ; First store ASCII minus sign.
0E                  STR   R2,R1,#0
0F                  NOT   R0,R0         ; Convert the number to absolute
10                  ADD   R0,R0,#1      ; value; it is easier to work with.
11   ;
12   Begin100       LD    R2,ASCIIoffset ; Prepare for "hundreds" digit.
13   ;
14                  LD    R3,Neg100     ; Determine the hundreds digit.
15   Loop100        ADD   R0,R0,R3
16                  BRn   End100
17                  ADD   R2,R2,#1
18                  BRnzp Loop100
19   ;
1A   End100         STR    R2,R1,#1    ; Store ASCII code for hundreds digit.
1B                  LD     R3,Pos100
1C                  ADD    R0,R0,R3    ; Correct R0 for one-too-many subtracts.
1D   ;
1E                  LD     R2,ASCIIoffset ; Prepare for "tens" digit.
1F   ;
20   Begin10        LD     R3,Neg10    ; Determine the tens digit.
21   Loop10         ADD    R0,R0,R3
22                  BRn    End10
23                  ADD    R2,R2,#1
24                  BRnzp  Loop10
25   ;
26   End10          STR    R2,R1,#2    ; Store ASCII code for tens digit.
27                  ADD    R0,R0,#10   ; Correct R0 for one-too-many subtracts.
29   Begin1         LD     R2,ASCIIoffset ; Prepare for "ones" digit.
2A                  ADD    R2,R2,R0
2B                  STR    R2,R1,#3
2C                  RET
2D   ;
2E   ASCIIplus      .FILL  x002B
2F   ASCIIminus     .FILL  x002D
30   ASCIIoffset    .FILL  x0030
31   Neg100         .FILL  xFF9C
32   Pos100         .FILL  x0064
33   Neg10          .FILL  xFFF6
```

Figure 10.20 Binary-to-ASCII conversion routine

10.5 Our Final Example: The Calculator

We conclude Chapter 10 with the code for a comprehensive example: the simulation of a calculator. The intent is to demonstrate the use of many of the concepts discussed thus far, as well as to show an example of well-documented, clearly written code, where the example is much more complicated than what can fit on one or two pages. The calculator simulation consists of 11 separate routines. You are encouraged to study this example before moving on to Chapter 11 and High-Level Language Programming.

The calculator works as follows: We use the keyboard to input commands and decimal values. We use the monitor to display results. We use a stack to perform arithmetic operations as described in Section 10.2. Values entered and displayed are restricted to three decimal digits, that is, only values between −999 and +999, inclusive. The available operations are

X Exit the simulation.

D Display the value at the top of the stack.

C Clear all values from the stack.

+ Replace the top two elements on the stack with their sum.

***** Replace the top two elements on the stack with their product.

- Negate the top element on the stack.

Enter Push the value typed on the keyboard onto the top of the stack.

Figure 10.21 is a flowchart that gives an overview of our calculator simulation. Simulation of the calculator starts with initialization, which includes setting R6, the stack pointer, to an empty stack. Then the user sitting at the keyboard is prompted for input.

Input is echoed, and the calculator simulation systematically tests the character to determine the user's command. Depending on the user's command, the calculator simulation carries out the corresponding action, followed by a prompt for another command. The calculator simulation continues in this way until the user presses X, signaling that the user is finished with the calculator.

Eleven routines comprise the calculator simulation. Figure 10.22 is the main algorithm. Figure 10.23 takes an ASCII string of digits typed by a user, converts it to a binary number, and pushes the binary number onto the top of the stack. Figure 10.19 provides the ASCII-to-binary conversion routine. Figure 10.26 pops the entry on the top of the stack, converts it to an ASCII string, and displays the ASCII string on the monitor. Figure 10.20 provides the binary-to-ASCII conversion routine. Figures 10.10 (OpAdd), 10.14 (OpMult), and 10.15 (OpNeg) supply the basic arithmetic algorithms using a stack. Figures 10.24 and 10.25 contain versions of the POP and PUSH routines tailored for this application. Finally Figure 10.27 clears the stack.

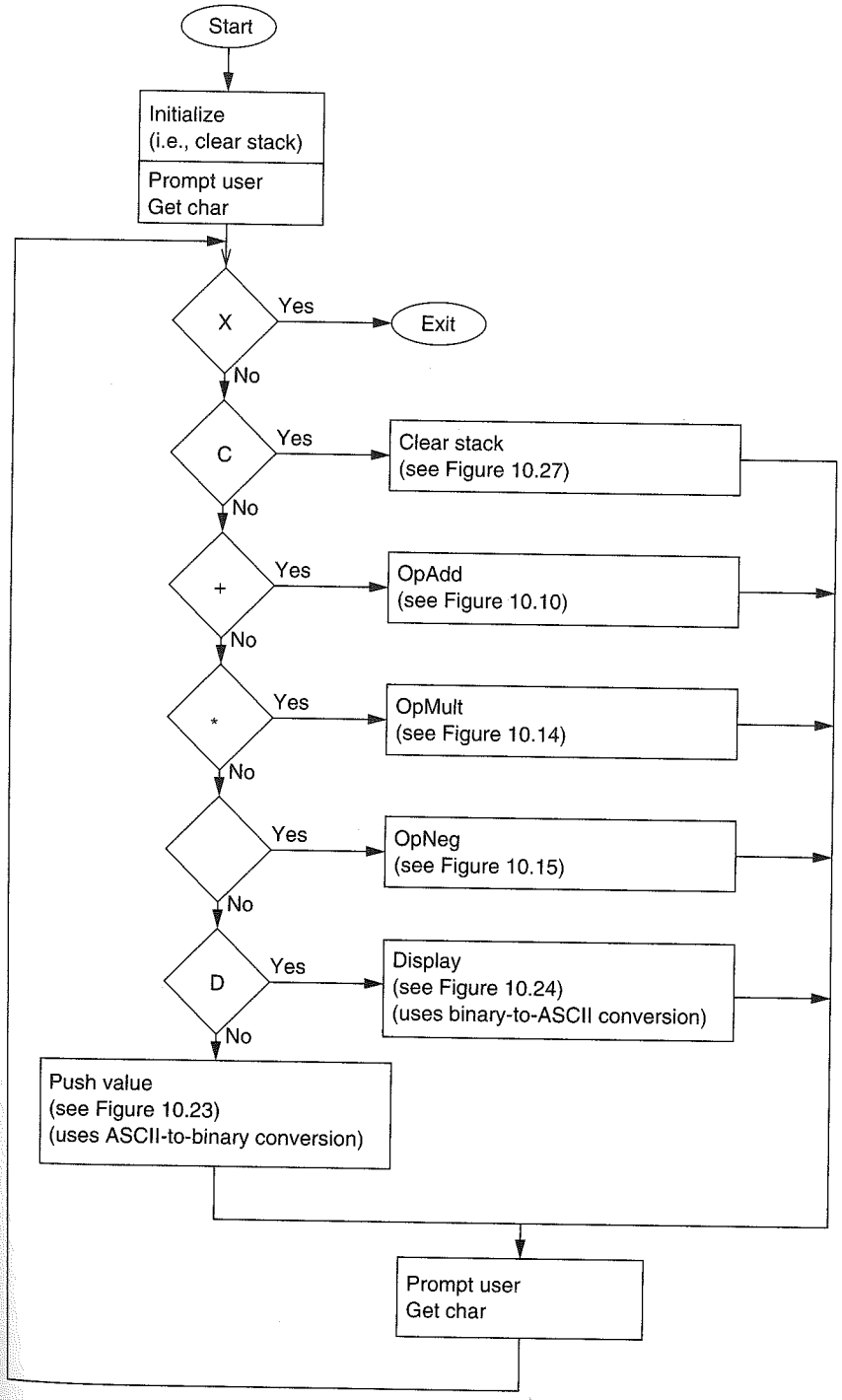


Figure 10.21 The calculator, overview

```
01    ;
02    ;  The Calculator, Main Algorithm
03    ;
04                LEA     R6,StackBase  ; Initialize the stack.
05                ADD     R6,R6,#-1     ; R6 is stack pointer.
06                LEA     R0,PromptMsg
07                PUTS
08                GETC
09                OUT
0A    ;
0B    ; Check the command
0C    ;
0D    Test        LD      R1,NegX       ; Check for X.
0E                ADD     R1,R1,R0
0F                BRz     Exit
10    ;           LD      R1,NegC       ; Check for C.
11                ADD     R1,R1,R0
12                BRz     OpClear       ; See Figure 10.27.
13
14    ;           LD      R1,NegPlus    ; Check for +
15                ADD     R1,R1,R0
16                BRz     OpAdd         ; See Figure 10.10.
17
18    ;           LD      R1,NegMult    ; Check for *
19                ADD     R1,R1,R0
1A                BRz     OpMult        ; See Figure 10.14.
1B
1C    ;           LD      R1,NegMinus   ; Check for -
1D                ADD     R1,R1,R0
1E                BRz     OpNeg         ; See Figure 10.15.
1F
20    ;           LD      R1,NegD       ; Check for D
21                ADD     R1,R1,R0
22                BRz     OpDisplay     ; See Figure 10.24.
23
24    ;
25    ; Then we must be entering an integer
26    ;
27                BRnzp   PushValue     ; See Figure 10.23.
28    ;
29    NewCommand  LEA     R0,PromptMsg
2A                PUTS
2B                GETC
2C                OUT
2D                BRnzp   Test
2E    Exit        HALT
2F    PromptMsg   .FILL   x000A
30                .STRINGZ "Enter a command:"
31    NegX        .FILL   xFFA8
32    NegC        .FILL   xFFBD
33    NegPlus     .FILL   xFFD5
34    NegMinus    .FILL   xFFD3
35    NegMult     .FILL   xFFD6
36    NegD        .FILL   xFFBC
```

Figure 10.22 The calculator's main algorithm

```
01   ; This algorithm takes a sequence of ASCII digits typed by the user,
02   ; converts it into a binary value by calling the ASCIItoBinary
03   ; subroutine, and pushes the binary value onto the stack.
04   ;
05   PushValue          LEA       R1,ASCIIBUFF   ; R1 points to string being
06                      LD        R2,MaxDigits   ; generated.
07   ;
08   ValueLoop          ADD       R3,R0,xFFF6    ; Test for carriage return.
09                      BRz       GoodInput
0A                      ADD       R2,R2,#0
0B                      BRz       TooLargeInput
0C                      ADD       R2,R2,#-1      ; Still room for more digits.
0D                      STR       R0,R1,#0       ; Store last character read.
0E                      ADD       R1,R1,#1
0F                      GETC
10                      OUT                      ; Echo it.
11                      BRnzp     ValueLoop
12   ;
13   GoodInput          LEA       R2,ASCIIBUFF
14                      NOT       R2,R2
15                      ADD       R2,R2,#1
16                      ADD       R1,R1,R2       ; R1 now contains no. of char.
17                      JSR       ASCIItoBinary
18                      JSR       PUSH
19                      BRnzp     NewCommand
1A   ;
1B   TooLargeInput      GETC                     ; Spin until carriage return.
1C                      OUT
1D                      ADD       R3,R0,xFFF6
1E                      BRnp      TooLargeInput
1F                      LEA       R0,TooManyDigits
20                      PUTS
21                      BRnzp     NewCommand
22   TooManyDigits      .FILL     x000A
23                      .STRINGZ  "Too many digits"
24   MaxDigits          .FILL     x0003
```

Figure 10.23 The calculator's PushValue routine

Note that a few changes are needed if the various routines are to work with the main program of Figure 10.17. For example, OpAdd, OpMult, and OpNeg must all terminate with

```
BRnzp NewCommand
```

instead of RET. Also, some labels are used in more than one subroutine. If the subroutines are assembled separately and certain labels are identified as .EXTERNAL (see Section 9.2.5), then the use of the same label in more than one subroutine is not a problem. However, if the entire program is assembled as a single module, then duplicate labels are not allowed. In that case, one must rename some of the labels (e.g., Restore1, Restore2, Exit, and Save) so that all labels are unique.

```
01   ;  This algorithm POPs a value from the stack and puts it in
02   ;  R0 before returning to the calling program.  R5 is used to
03   ;  report success (R5 = 0) or failure (R5 = 1) of the POP operation.
04   POP             LEA     R0,StackBase
05                   NOT     R0,R0
06                   ADD     R0,R0,#2        ; R0 = -(addr.ofStackBase -1)
07                   ADD     R0,R0,R6        ; R6 = StackPointer
08                   BRz     Underflow
09                   LDR     R0,R6,#0        ; The actual POP
0A                   ADD     R6,R6,#1        ; Adjust StackPointer
0B                   AND     R5,R5,#0        ; R5 <-- success
0C                   RET
0D   Underflow       ST      R7,Save         ; TRAP/RET needs R7.
0E                   LEA     R0,UnderflowMsg
0F                   PUTS                    ; Print error message.
10                   LD      R7,Save         ; Restore R7.
11                   AND     R5,R5,#0
12                   ADD     R5,R5,#1        ; R5 <-- failure
13                   RET
14   Save            .FILL   x0000
15   StackMax        .BLKW   9
16   StackBase       .FILL   x0000
17   UnderflowMsg    .FILL   x000A
18                   .STRINGZ "Error: Too Few Values on the Stack."
```

Figure 10.24 The calculator's POP routine

```
01   ;  This algorithm PUSHes on the stack the value stored in R0.
02   ;  R5 is used to report success (R5 = 0) or failure (R5 = 1) of
03   ;  the PUSH operation.
04   ;
05   PUSH            ST      R1,Save1        ; R1 is needed by this routine.
06                   LEA     R1,StackMax
07                   NOT     R1,R1
08                   ADD     R1,R1,#1        ; R1 = - addr. of StackMax
09                   ADD     R1,R1,R6        ; R6 = StackPointer
0A                   BRz     Overflow
0B                   ADD     R6,R6,#-1       ; Adjust StackPointer for PUSH.
0C                   STR     R0,R6,#0        ; The actual PUSH
0D                   BRnzp   Success_exit
0E   Overflow        ST      R7,Save
0F                   LEA     R0,OverflowMsg
10                   PUTS
11                   LD      R7,Save
12                   LD      R1, Save1       ; Restore R1.
13                   AND     R5,R5,#0
14                   ADD     R5,R5,#1        ; R5 <-- failure
15                   RET
16   Success_exit    LD      R1,Save1        ; Restore R1.
17                   AND     R5,R5,#0        ; R5 <-- success
18                   RET
19   Save            .FILL   x0000
1A   Save1           .FILL   x0000
1B   OverflowMsg     .STRINGZ "Error: Stack is Full."
```

Figure 10.25 The calculator's PUSH routine

```
01   ; This algorithm calls BinarytoASCII to convert the 2's complement
02   ; number on the top of the stack into an ASCII character string, and
03   ; then calls PUTS to display that number on the screen.
04   OpDisplay   JSR      POP          ; R0 gets the value to be displayed.
05               ADD      R5,R5,#0
06               BRp      NewCommand   ; POP failed, nothing on the stack.
07               JSR      BinarytoASCII
08               LD       R0,NewlineChar
09               OUT
0A               LEA      R0,ASCIIBUFF
0B               PUTS
0C               ADD      R6,R6,#-1    ; Push displayed number back on stack.
0D               BRnzp    NewCommand
0E   NewlineChar .FILL    x000A
```

Figure 10.26 The calculator's display routine

```
01   ;
02   ; This routine clears the stack by resetting the stack pointer (R6).
03   ;
04   OpClear     LEA      R6,StackBase  ; Initialize the stack.
05               ADD      R6,R6,#1      ; R6 is stack pointer.
06               BRnzp    NewCommand
```

Figure 10.27 The OpClear routine

Exercises

10.1 What are the defining characteristics of a stack?

10.2 What is an advantage to using the model in Figure 10.3 to implement a stack versus the model in Figure 10.2?

10.3 The LC-3 ISA has been augmented with the following Push and Pop instructions. Push Rn pushes the value in Register n onto the stack. Pop Rn removes a value from the stack and loads it into Rn. The figure below shows a snapshot of the eight registers of the LC-3 BEFORE and AFTER the following six stack operations are performed. Identify (a)–(d).

	BEFORE
R0	x0000
R1	x1111
R2	x2222
R3	x3333
R4	x4444
R5	x5555
R6	x6666
R7	x7777

```
PUSH R4
PUSH (a)
POP  (b)
PUSH (c)
POP  R2
POP  (d)
```

	AFTER
R0	x1111
R1	x1111
R2	x3333
R3	x3333
R4	x4444
R5	x5555
R6	x6666
R7	x4444

10.4 Write a function that implements another stack function, peek. Peek returns the value of the first element on the stack without removing the element from the stack. Peek should also do underflow error checking. (Why is overflow error checking unnecessary?)

10.5 How would you check for underflow and overflow conditions if you implemented a stack using the model in Figure 10.2? Rewrite the PUSH and POP routines to model a stack implemented as in Figure 10.2, that is, one in which the data entries move with each operation.

10.6 Rewrite the PUSH and POP routines such that the stack on which they operate holds elements that take up two memory locations each.

10.7 Rewrite the PUSH and POP routines to handle stack elements of arbitrary sizes.

10.8 The following operations are performed on a stack:

```
PUSH A, PUSH B, POP, PUSH C, PUSH D, POP, PUSH E,
POP, POP, PUSH F
```

a. What does the stack contain after the `PUSH F`?

b. At which point does the stack contain the most elements? Without removing the elements left on the stack from the previous operations, we perform:

```
PUSH G, PUSH H, PUSH I, PUSH J, POP, PUSH K,
POP, POP, POP, PUSH L, POP, POP, PUSH M
```

c. What does the stack contain now?

10.9 The input stream of a stack is a list of all the elements we pushed onto the stack, in the order that we pushed them. The input stream from Exercise 10.8 was `ABCDEFGHIJKLM`

The output stream is a list of all the elements that are popped off the stack, in the order that they are popped off.

a. What is the output stream from Exercise 10.8?
Hint: BDE . . .

b. If the input stream is ZYXWVUTSR, create a sequence of pushes and pops such that the output stream is YXVUWZSRT.

c. If the input stream is ZYXW, how many different output streams can be created?

10.10 During the initiation of the interrupt service routine, the N, Z, and P condition codes are saved on the stack. Show by means of a simple example how incorrect results would be generated if the condition codes were not saved.

10.11 In the example of Section 10.2.3, what are the contents of locations x01F1 and x01F2? They are part of a larger structure. Provide a name for that structure. (*Hint:* See Table A.3.)

10.12 Expand the example of Section 10.2.3 to include an interrupt by a still more urgent device D while the service routine of device C is executing the instruction at x6310. Assume device D's interrupt vector is xF3. Assume the interrupt service routine is stored in locations x6400 to x6412. Show the contents of the stack and PC at each relevant point in the execution flow.

10.13 Suppose device D in Exercise 10.12 has a lower priority than device C but a higher priority than device B. Rework Exercise 10.12 with this new wrinkle.

10.14 Write an interrupt handler to accept keyboard input as follows: A buffer is allocated to memory locations x4000 through x40FE. The interrupt handler must accept the next character typed and store it in the next "empty" location in the buffer. Memory location x40FF is used as a pointer to the next available empty buffer location. If the buffer is full (i.e., if a character has been stored in location x40FE), the interrupt handler must display on the screen: "Character cannot be accepted; input buffer full."

10.15 Consider the interrupt handler of Exercise 10.14. The buffer is modified as follows: The buffer is allocated to memory locations x4000 through x40FC. Location x40FF contains, as before, the address of the next available empty location in the buffer. Location x40FE contains the address of the oldest character in the buffer. Location x40FD contains the number of characters in the buffer. Other programs can remove characters from the buffer. Modify the interrupt handler so that, after x40FC is filled, the next location filled is x4000, assuming the character in x4000 has been previously removed. As before, if the buffer is full, the interrupt handler must display on the screen: "Character cannot be accepted; input buffer full."

10.16 Consider the modified interrupt handler of Exercise 10.15, used in conjunction with a program that removes characters from the buffer. Can you think of any problem that might prevent the interrupt handler that is adding characters to the buffer and the program that is removing characters from the buffer from working correctly together?

10.17 Describe, in your own words, how the Multiply step of the OpMult algorithm in Figure 10.14 works. How many instructions are executed to perform the Multiply step? Express your answer in terms of n, the value of the multiplier. (*Note:* If an instruction executes five times, it contributes 5 to the total count.) Write a program fragment that performs the Multiply step in fewer instructions if the value of the multiplier is less than 25. How many?

10.18 Correct Figure 10.16 so that it will add two single-digit positive integers and produce a single-digit positive sum. Assume that the two digits being added do in fact produce a single-digit sum.

10.19 Modify Figure 10.16, assuming that the input numbers are one-digit positive hex numbers. Assume that the two hex digits being added together do in fact produce a single hex-digit sum.

10.20 Figure 10.19 provides an algorithm for converting ASCII strings to binary values. Suppose the decimal number is arbitrarily long. Rather than store a table of 10 values for the thousands-place digit, another table for the 10 ten-thousands-place digit, and so on, design an algorithm to do the conversion without resorting to any tables whatsoever.

10.21 The code in Figure 10.19 converts a decimal number represented as ASCII digits into binary. Extend this code to also convert a hexadecimal number represented in ASCII into binary. If the number is preceded by an x, then the subsequent ASCII digits (three at most) represent a hex number; otherwise it is decimal.

10.22 The algorithm of Figure 10.20 always produces a string of four characters independent of the sign and magnitude of the integer being converted. Devise an algorithm that eliminates unnecessary characters in common representations, that is, an algorithm that does not store leading 0s nor a leading + sign.

10.23 What does the following LC-3 program do?

```
            .ORIG    X3000
            LEA      R6, STACKBASE
            LEA      R0, PROMPT
            TRAP     x22             ; PUTS
            AND      R1, R1, #0
LOOP        TRAP     x20             ; IN
            TRAP     x21
            ADD      R3, R0, #-10    ; Check for newline
            BRz      INPUTDONE
            JSR      PUSH
            ADD      R1, R1, #1
            BRnzp    LOOP
INPUTDONE   ADD      R1, R1, #0
            BRz      DONE
LOOP2       JSR      POP
            TRAP     x21
            ADD      R1, R1, #-1
            BRp      LOOP2
DONE        TRAP     x25             ; HALT

PUSH        ADD      R6, R6, #-2
            STR      R0, R6, #0
            RET

POP         LDR      R0, R6, #0
            ADD      R6, R6, #2
            RET
PROMPT      .STRINGZ ''Please enter a sentence: ''
STACKSPAC   .BLKW #50
STACKBASE   .FILL #0
            .END
```

10.24 Suppose the keyboard interrupt vector is x34 and the keyboard interrupt service routine starts at location x1000. What can you infer about the contents of any memory location from the above statement?